LEVEL

12

AD A068863

Contract Period Covered by Report: 1 September 1978 – 30 November 1978

9 Quarterly research and development technical rept. 1 Sep-30 Nov 78,

11 30 Nov 78

12 83 p.

Quarterly Research and Development Technical Report

6 Spatial Data Management System,

Computer Corporation of America

The views and conclusions in this document are those of the authors and should not be interpreted as necessarily representing the official policies, express or implied, of the Advanced Research Projects Agency, or the United States Government.

Report Authors: 10 Christopher F. Herot, David Kramlich, Richard Carling, Mark Friedell, Jerry Farrell

~~Research Division~~
Computer Corporation of America
617-491-3670

Sponsor: Defense Advanced Research Projects Agency, Office of Cybernetics Technology

ARPA Order Number: 3487

ARPA Contract Number: 15 MDA903-78-C-0122, ARPA Order-3487

Contract Period: ~~15 February 1978~~ – 30 November 1979

Table of Contents

1. INTRODUCTION

This fourth quarter of work on the design and implementation of a prototype Spatial Data Management System (SDMS) resulted in the first operational version of the system. The bulk of this effort was devoted to constructing the mechanism through which a user can create, modify, and view an image plane.

An image plane is a flat surface upon which a user can store information. As that surface may be considerably larger than the display screen, the SDMS provides a window which can be moved over that surface in order to view parts of it. As the window is moved, the window's position is indicated on an auxiliary screen which serves as a navigational aid containing a map of the entire image plane.

This implementation of SDMS at CCA is the first such system to integrate the I-Space creation and viewing operations so that a user can create and modify a database without the need of a skilled computer professional.

Chapter 2 describes the hardware configuration used in the prototype, with special emphasis being placed on the display devices.

Chapter 3 describes the motion control mechanisms which allow the user to maneuver over an Information Space.

Chapter 4 describes the programs which allow a user to create and modify an Information Space.

Chapter 5 summarizes the work to date on the programs for creating Information Spaces from symbolic data stored in the INGRES relational data base.

Appendix A contains the module descriptions from the Detailed Design Document [HEROT et al.] which pertain to the work accomplished this quarter.

Finally, Appendix B describes those statements of Icon Class Description Language (ICDL) which have been implemented this quarter.

Plate 1   SDMS User Station

The monitor at left displays the navigational aid and menu.
The monitor at right displays the Information Space.
The tablet and joystick are in the foreground.

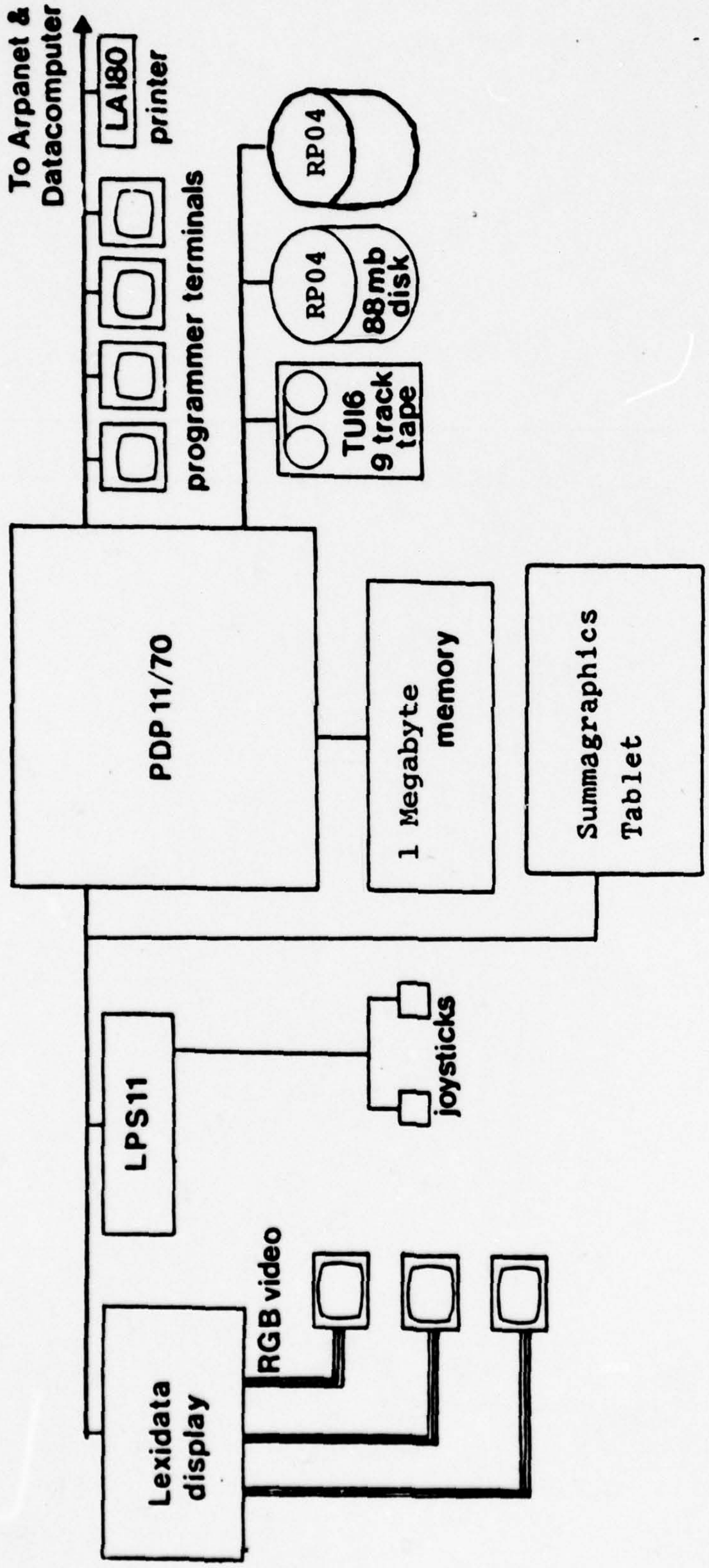
2. HARDWARE CONFIGURATION

The prototype CCA Spatial Data Management System is designed to provide an individual user with graphical representations of both private and shared databases. It employs a dedicated PDP-11/70 processor with a megabyte of memory and a variety of peripherals (see Figure 2.1). The most important of these is the Lexidata display, which provides the user with a view of the Information Space and various navigational aids.

2.1 Lexidata Display

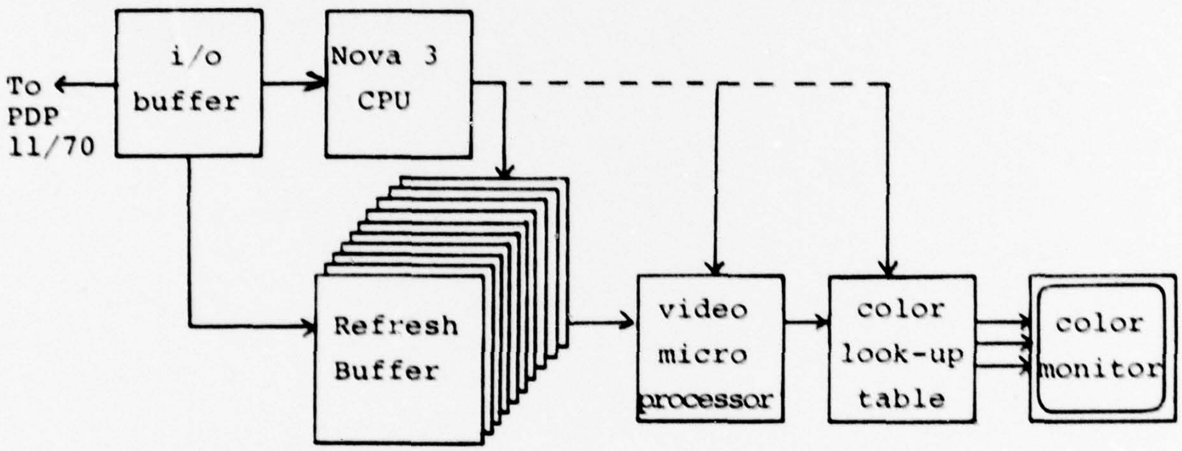
The Lexidata 6400 is typical of the current generation of raster scan displays, although it has several valuable features which make it especially suitable for use with SDMS. It is a frame buffer, that is, it employs a memory which stores the individual picture elements (pixels) of an image in discrete memory cells. There are 311,680 such cells, providing for 487 lines of 640 pixels each. The contents of each cell are accessed as the electron beam scans the corresponding location on the display tube, a process which is repeated 30 times each second. Each memory cell consists of nine bits. Typically, eight of

SDMS Hardware Configuration | Figure 2.1



these are used to specify the color of the corresponding pixel on the screen. The ninth bit is used to provide overlay functions such as cursors. The architecture of the currently installed Lexidata is shown in Figure 2.2.

Lexidata Architecture   Figure 2.2



While many raster scan displays rely on hard-wired logic to perform the refresh operation, the Lexidata uses a programmable video microprocessor. This feature provides the flexibility required to achieve the smooth scrolling and zooming effects required by SDMS.

The Lexidata also contains a Nova minicomputer which performs control operations and generates graphics (such as vectors and conics) in the frame buffer. Data can also be stored in the frame buffer via a high speed parallel connection to the PDP-11 interface.

In January, the currently installed Lexidata will be replaced by a newer version which has several important new features:

1. three independent color display channels
2. zooming in integer steps from 1 to 16
3. horizontal scrolling in single memory-pixel steps
4. vertical scrolling with wrap-around
5. fast updates of vertical columns of pixels
6. ability to synchronize to an external video signal

The first of these features will allow driving three CRTs from the one display system, in effect providing three color displays in one chasis. One of the displays will have nine bits per point as in the current system. The other two will provide four bits per point on each display. Upon installation of this system, the Ramtek GX-100B will be retired. The Ramtek is old and is becoming increasingly expensive to maintain and is already prohibitively expensive to expand or replicate.

The second new feature, integer zooming, is essential for letting the user make transitions between the different

image planes which make up an Information Space in SDMS.

The third feature, horizontal scrolling, is already implemented at scale 2 in the current Lexidata, but is not operational at higher scales and results in motion of the margins of the displayed image. While these are currently hidden by masking the corresponding area of the monitor, the new feature will solve the problem at the video signal, allowing direct recording of the output.

A fourth improvement allows the video processor's addressing of the Lexidata memory to wrap-around at the end, permitting vertical scrolling over image plane higher than 480 lines.

A fifth important feature is a special mode to allow pixels in any arbitrary rectangle to be updates at 1 usec each, as long as the rectangle is aligned on a word boundary. This feature will reduce the time required to send the data from the PDP-11 to the display and allow smoother motion about the I-Space.

A sixth feature is the provision for synchronizing the video signal to an external source and encoding the output of the display for recording on video tape.

3. MOTION

A central component of SDMS is the facility which allows the user to move about the image plane upon which he stores his information. The user views these image planes through a window which he can maneuver. Currently, motion is implemented parallel to the image plane, and is controlled by pressing a joy stick in the direction in which the user wishes to travel.

The effect of motion is achieved by changing the portion of the image plane which is displayed on the user's CRT. As the frame buffer is not sufficiently large to contain an entire image plane, this must be accomplished by moving data from the disk, where the entire image plane is stored as an array of pixels, to the display. This process is referred to as staging and is described in detail in this section.

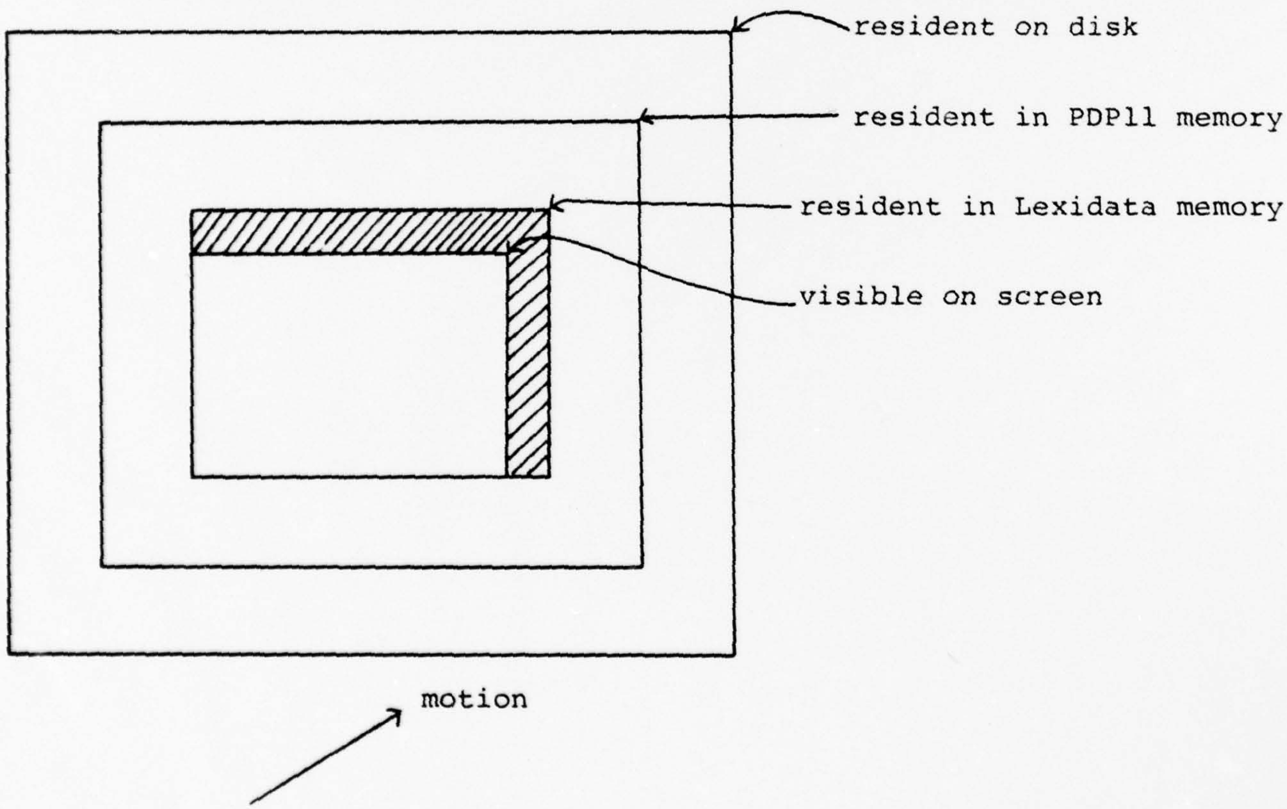
3.1 Storage Levels

The image planes of SDMS are stored at multiple levels in the system. The data appearing on the screen, together with a small amount of the surrounding area, is stored in the Lexidata frame buffer. A somewhat larger area is

stored in the memory of the PDP-11/70. The image plane is stored in its entirety on an RP04 88Mb moving head disk. Eventually, compressed representations of image planes may be stored at other locations on the Arpanet, such as the datacomputer. The nesting of these representations is illustrated in Figure 3.1. Motion across an image plane requires staging data from the disk to the PDP-11's memory

---

Nesting of Data Figure 3.1



---

to the Lexidata in order to maintain the required nesting.

3.1.1 Tiles on the disk

The image plane is broken into tiles for storage on the moving head disk. The size and distribution of tiles on the disk have been designed to minimize the amount of data that must be read into primary memory to satisfy motion in any given direction, and ensure that the time required to move the head to the appropriate tiles and read them in will be equal for any direction of motion over the image plane. The optimal tile size, determined with the aid of a simulation, was 128 pixels wide by 64 lines high for a screen with standard (4:3) aspect ration. If motion were limited by disk time alone, such a storage scheme would allow one 640 x 480 screenful of data to be moved in the worst direction (diagonally) in about 3 seconds; at the more common scale 2 (320 x 240), the time would be about 0.8 seconds.

3.1.2 Tiles in core

The system attempts to keep all tiles containing the information currently on the screen, plus a suitable margin, in core at all times. The addresses of these tiles are stored in an array called the <u>tile map</u>. This array allows any reference to a point on the image plane to be mapped into the appropriate address in the core buffer.

Along with the address of the tile, each element of the tile map contains the disk address where the tile is stored and a number of flag bits which indicate whether the data in the core resident tile is valid and whether it has been modified.

The tile map allows the motion control system to ensure that a suitable margin of data is maintained in core at all times. When scrolling has caused the displayed area to deviate from the center of the data represented by the tile map, the table is <u>rolled</u> -- that is, the data in it is shuffled so that the data represented in the center of the tile map is once again that which is in the center of the screen. This process is illustrated for the case of

---

Rolling of Tile Map Figure 3.2

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49
50	51	52	53	54	55	56

2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36
37	38	39	40	41	42	43
44	45	46	47	48	49	50
51	52	53	54	55	56	57

motion

---

motion to the right in Figure 3.2. The tile addresses which have wrapped around from one margin to the other, shown as the shaded column in the figure, are flagged as being invalid. The motion control system then computes

the disk addresses necessary to fill these tiles with new data corresponding to the new margin and issues the appropriate disk read requests. The disk reading process flags each tile as containing valid data when it has been read in.

The tile map is also used by programs which modify the image on the screen and/or on the disk. Both the Graphical Data Space editor, described in Section 4, and the icon generation programs, described in Section 5, use the tile map to determine which core locations must be modified to perform a primitive image creation function. After modifying the appropriate image data, such a program sets the corresponding modification bits in the tile map entry, causing the system to write the modified tiles back to the disk when they are rolled out of the map.

A program may make additions to the image plane without setting the modification bit. Such a technique will be used for the output of the BLINK and FRAME statements, which will cause icons to be modified while they are on the screen but will not change them permanently on the disk. In this case, the modifications must be repeated each time a tile is read in from the disk.

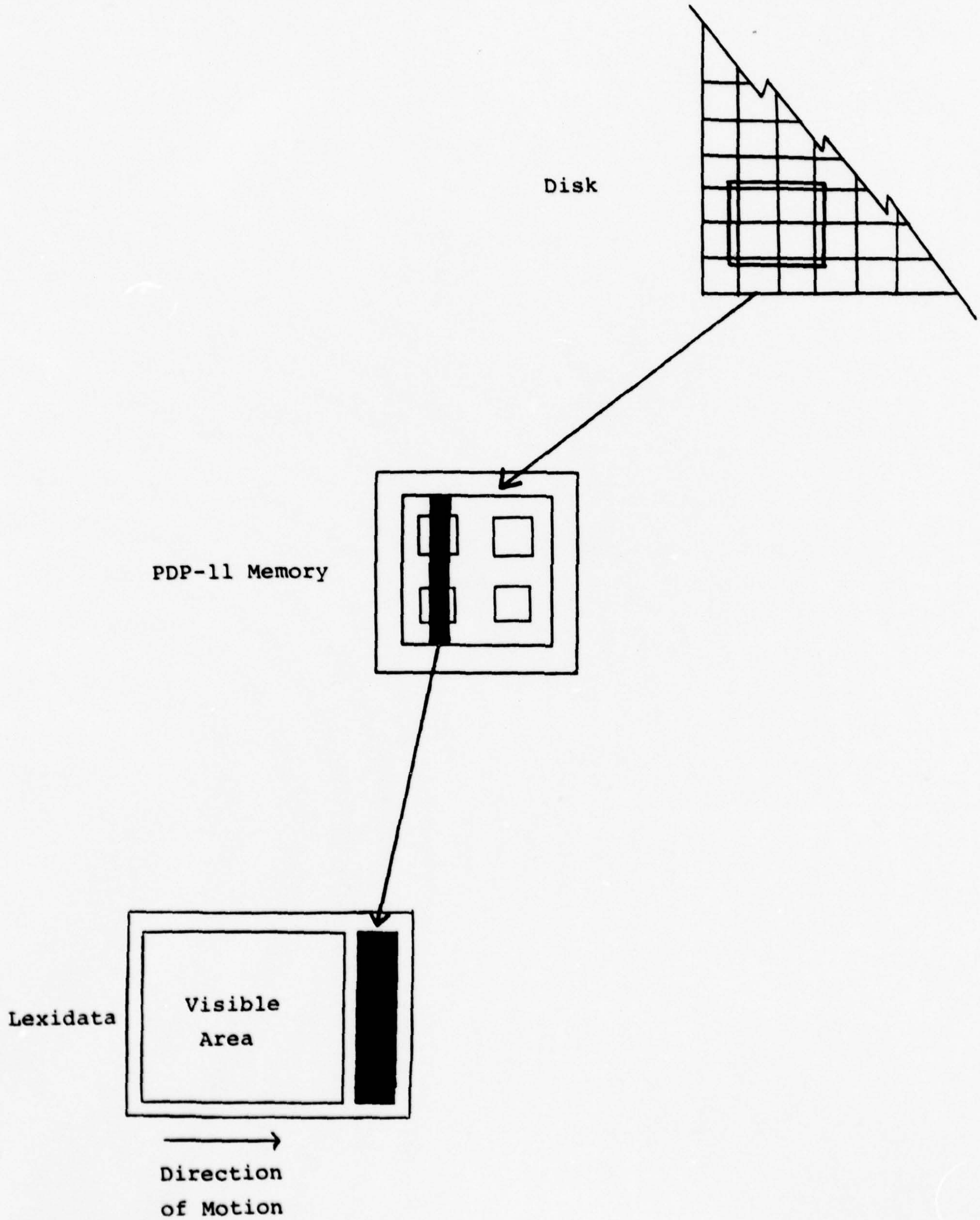
3.1.3 Data in the display

In order to be seen on the CRT, image plane data must be resident in the Lexidata display. The system endeavors to maintain a margin of data around the visible image. As the user causes the display to scroll into this margin, the system re-uses the memory left behind, writing new image data into it and thus preparing a new margin.

The re-use of the Lexidata memory requires that the display refresh logic address the memory in such a way that the memory appears to "wrap-around" from line to line and from top to bottom. The data in the display is stored as a set of contiguous scan lines. This fact, together with the limited size of the display's memory, requires that only those portions of tiles which are visible, together with a small margin, be sent to the Lexidata from the PDP-11. These portions are composed of stripes of data which the motion software writes into the margin just ahead of the current direction of travel (see Figure 3.3).

Data Staging for Scrolling

Figure 3.3



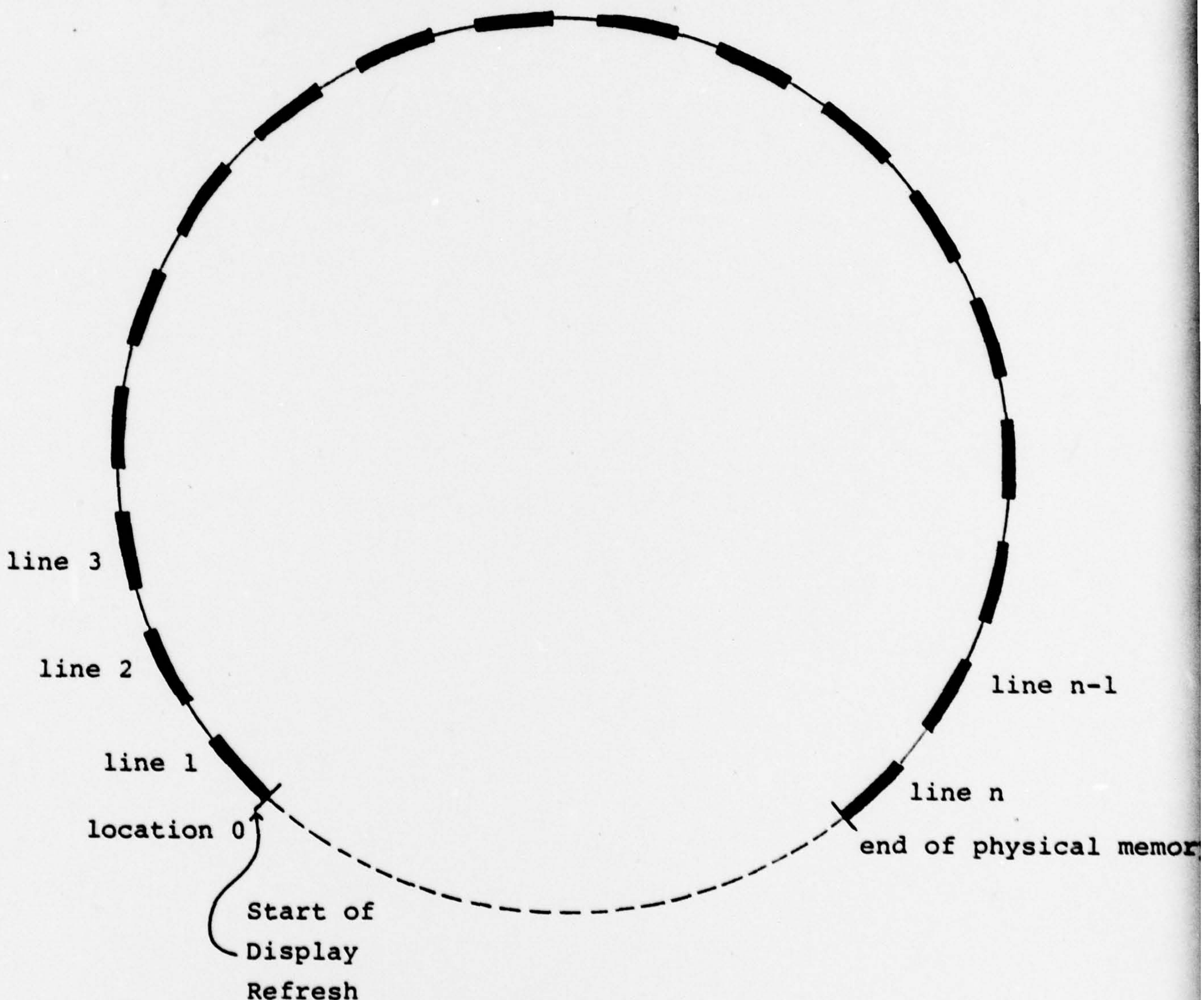
3.2 Scrolling

The operation most fundamental to SDMS is scrolling - the means by which the user manipulates his view of the image plane. Scrolling is performed by changing the point in the buffer at which the display refreshes the screen while simultaneously writing new data into that refresh buffer, producing the effect of continuous motion over a surface which may be many times larger than the refresh buffer itself. This process is best understood by visualizing the refresh buffer as a linear progression of memory cells, as illustrated in Figure 3.4. The state of the display when the hardware is initialized is shown in Figure 3.4(a). Starting at the point labeled "start of display refresh", the video output processor reads contiguous locations in memory until it has filled the screen. This action is repeated 30 times each second.

If the start of display pointer is incremented by one line width, so that the display starts at the second line in memory, as shown in Figure 3.4(b), that second line in memory will now be the first line to be displayed on the screen, causing a scroll in the vertical direction. Note that the display refresh will reach the end of memory

Display in initial state

Figure 3.4(a)



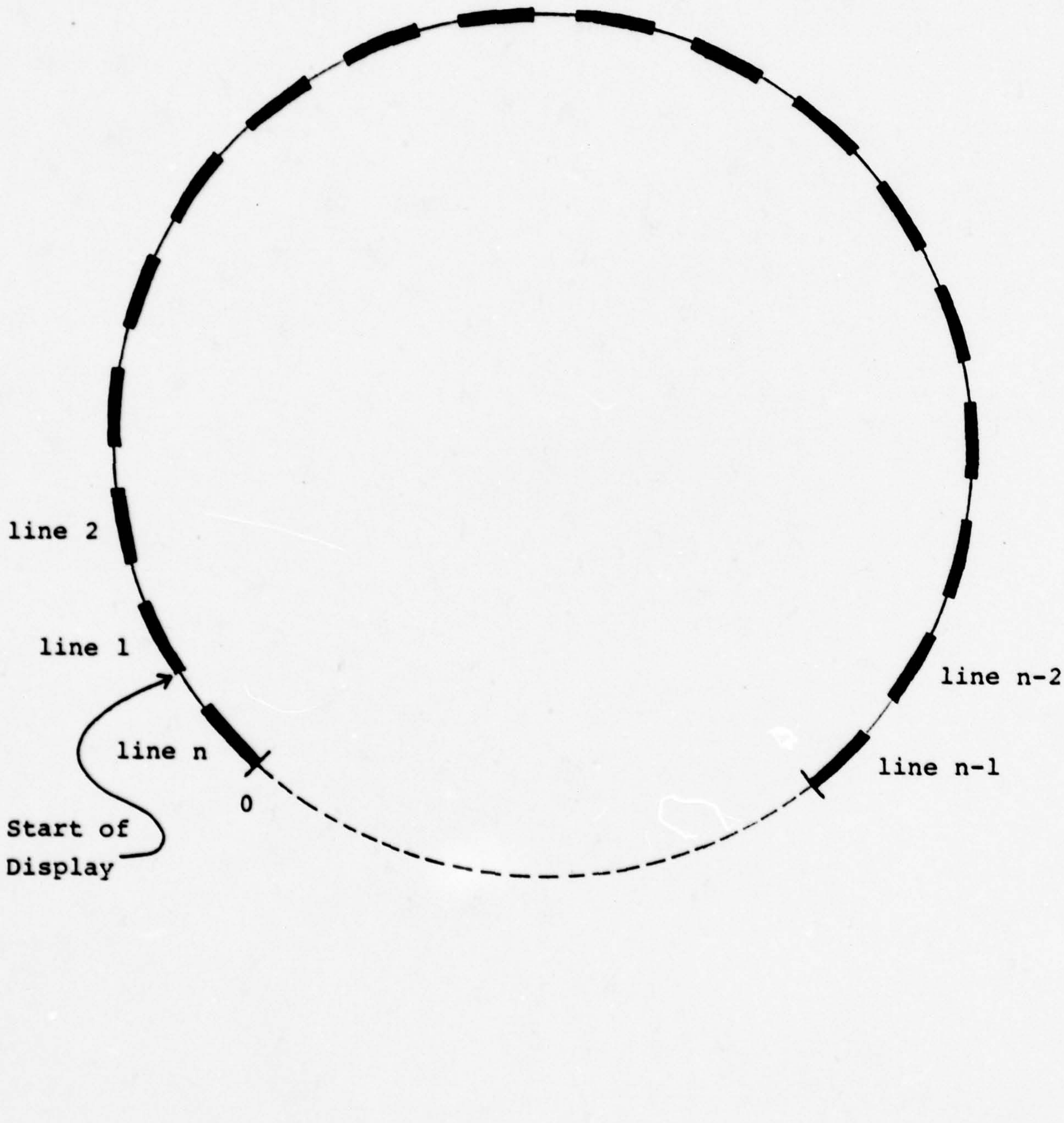
before it reaches the bottom of the screen. How it chooses the next line determines whether scrolling can be repeated indefinitely or not. To achieve continuous scrolling, the refresh processor's memory addressing scheme must wrap around, such that once it reaches the end of memory, the refresh continues uninterrupted from the beginning of memory. If the address of the last word of physical memory is also the last address in the <u>address space</u>, that is:

last_memory_location=2**number_of_bits_in_an_address-1

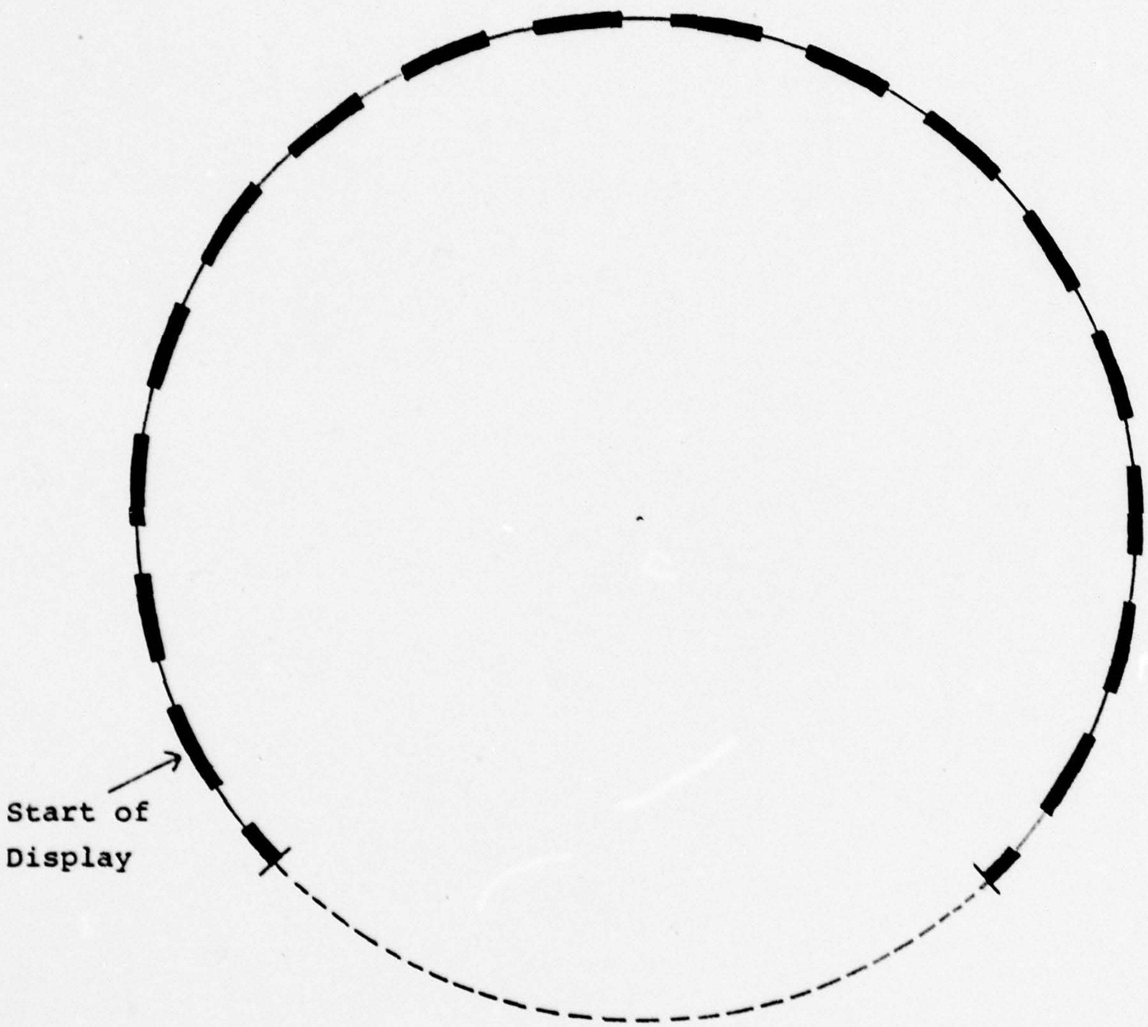
then the desired wrap-around follows directly from the memory design. Otherwise, the refresh processor must be capable of detecting the end of memory and restarting at the beginning.

Continuous scrolling also requires that new data be written into the refresh buffer. In order for this to be accomplished without any visible artifacts, some number of lines in the memory are reserved for use as a margin and not displayed. New data is written into this margin which, given the above specified wrap-around, is the area of memory immediately preceding the start of the display pointer. This new data is then scrolled onto the screen, freeing up a new portion of memory which, in turn, becomes the margin.

Display scrolled vertically by 1 line Figure 3.4(b)



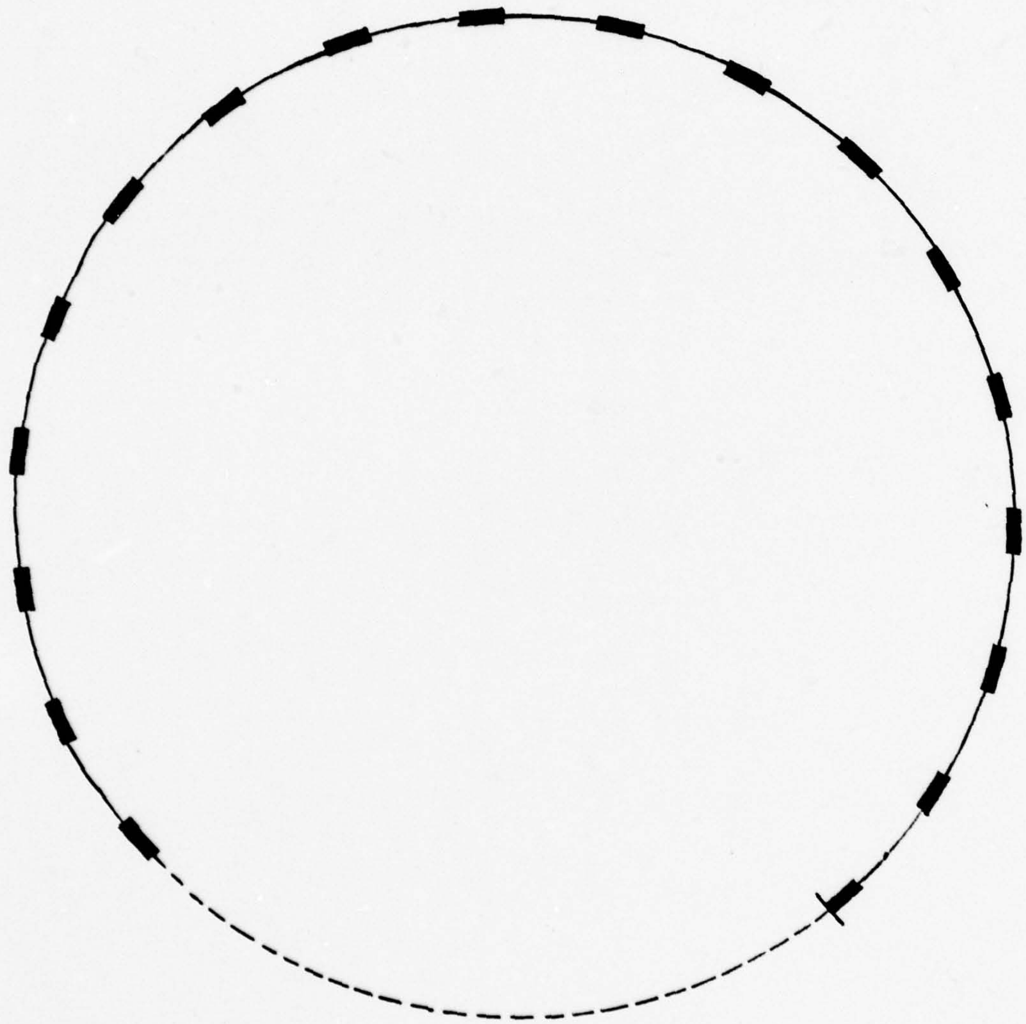
Display scrolled by .5 screen width Figure 3.4(c)



Horizontal scrolling is performed in a similar manner. Consider the effect of moving the start-of-display pointer by some fraction of a line, as shown in Figure 3.4(c). This has the effect of shifting every line of the display over to the left by that amount. Just as vertical scrolling results in data which was formerly at the bottom of the screen appearing at the top, horizontal scrolling results in data which was at the beginning of one line showing up at the end of the preceding line. If there is sufficient undisplayed space between lines (as shown by the gaps between the heavy lines in the figures) new data can be written into these gaps before they are scrolled onto the screen. These gaps are precisely the same as the margin depicted to the right of the screen in figure 3.3. Note that a display which has been scrolled horizontally by one screen width is in precisely the same state as one which has been scrolled vertically by one scan line. This phenomenon permits a significant degree of horizontal scrolling on displays without memory wrap-around, as only one scan line must be given up for every desired screen width of scroll. (The current Lexidata with 480 visible lines and 487 lines in memory thus permits scrolling over a space seven times as wide as it is high.)

Zooming the display can be illustrated by the same model. As shown in Figure 3.4(d) the display in the zoomed state

Display zoomed to scale 2 Figure 3.4(d)



uses line which are shorter and fewer in number, corresponding to the chosen zoom factor. At scale 2, there are half as many lines each half as long as at scale one. Such a display could be scrolled over an area which was two screen widths on a side without using any memory wrap-around. (In the current Lexidata, by re-using the bottom seven lines as described above, an area which was 2 screens high by 14 screens long could be traversed.)

3.2.1 Motion Programs

Scrolling is supervised by a program known as the navigator, which is responsible for five activities:

1. Sending the command to the Lexidata to update the point in its buffer at which it starts its display, thereby causing the image to scroll. This operation must be done at constant intervals if the scrolling is to appear even. It must also be done at least 30 times each second if the scrolling is to appear smooth.

2. Sending a command to the navigational aid telling it to update the position of the cursor.

3. Sending data to the Lexidata to maintain the margin of image data which the scroll command can move in

to. Enough data must be sent to ensure that the Lexidata can always have new data to scroll on the screen.

4. Causing new tiles to be read in from the disk so as to maintain enough data in the PDP-11's memory to be sent to the Lexidata when needed. Once again, this must be done fast enough so that there is always data to feed the Lexidata.

5. Maintaining status information (location, scale, coordinate mappings, etc.) for use by other processes, e.g. in interpreting queries relating to objects on the screen.

Individually, none of these activities requires an inordinate amount of time. Taken together, they present formidable problems if they are to be orchestrated in such a manner as to collectively meet all of the criteria set forth above. Accordingly, the solution required two iterations which are outlined below.

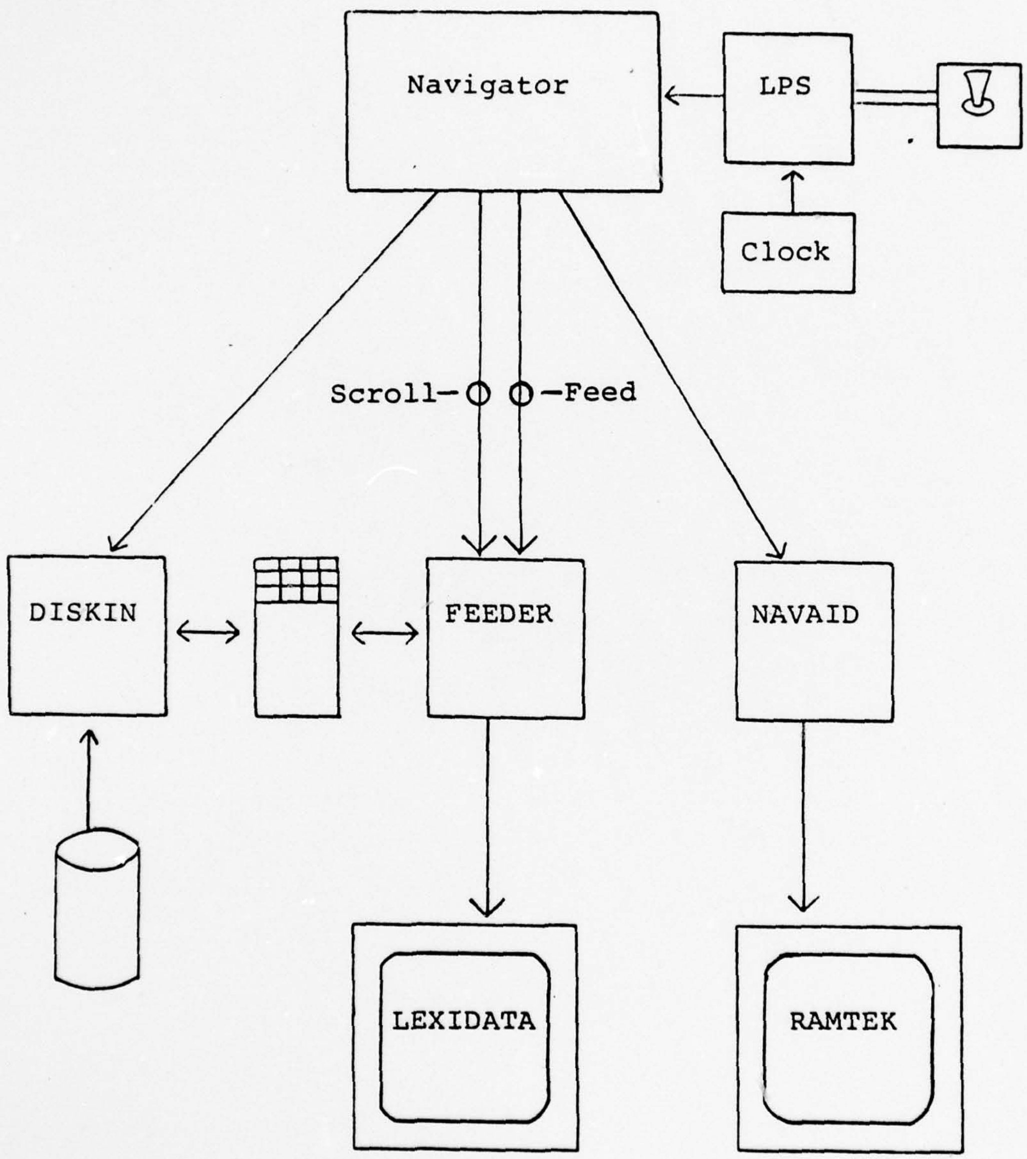
3.2.2 First implementation

The initial implementation of SDMS motion employed four processes communicating through pipes and shared memory. The four processes were (picture in Figure 3.5):

1. The navigator, which read the joy sticks, updated

Motion Processes

Figure 3.5



current status, and dispatched commands through pipes to the other processes.

2. The disk reader, which read tiles in from the disk.
3. The feeder which sent scroll requests and data to he Lexidata upon receiving commands from the navigator.
4. The navigational aid, which maintained the position marker on the world view map.

The tile map, the tiles themselves, and various parameters of the system were stored in a section of core accessable to all SDMS processes, eliminating the necessity of sending large quantities of data through pipes. The commands, however, were sent through pipes to achieve the necessary synchronization.

This approach served two purposes:

1. It allowed the four activities described above to take place asynchronously.
2. It kept the size of each program down to a level which would fit into the available address space of a PDP-11.

Unfortunately, such an elegant solution was also too inefficient. Motion though the image plane was slow and erratic. The problem was traced to the excessive overhead of using so many processes.

The navigator had two pipes for communicating with the feeder in order to allow scroll and feed requests to be processed in the proper order. While scroll requests were short and frequent (every 33 msec), feed requests tended to come in bunches. The intention was that the navigator would place these requests in the appropriate pipes as it generated them and then the feeder would process them, giving scroll requests the priority they required.

In practice, once the feeder had started to process the feed requests, there was no way of insuring that the navigator would be scheduled to run in time to read the joy stick before another frame time (33 msec) had elapsed. Inserting extra steps in the navigator-feeder pipe protocol to ensure that the navigator did run resulted in an inordinate time being spent in the Unix scheduler and its associated context switching. (Experiments indicated that each process switch via pipe i/o cost at least 5 ms.) In short, the four process approach had to be abandoned.

3.2.3 Second implementation

Reducing the number of processes in the SDMS motion facility required a modification to Unix to allow a single process to control more than one i/o operation. The original design of Unix did not permit a process which requested an

i/o operation to run again until that operation had been completed. This restriction freed the system's designers from providing for the eventuality that such a process would decide to terminate itself while an i/o request was still pending, a situation which could result in some data being read into core after that core had been re-assigned to some new process.

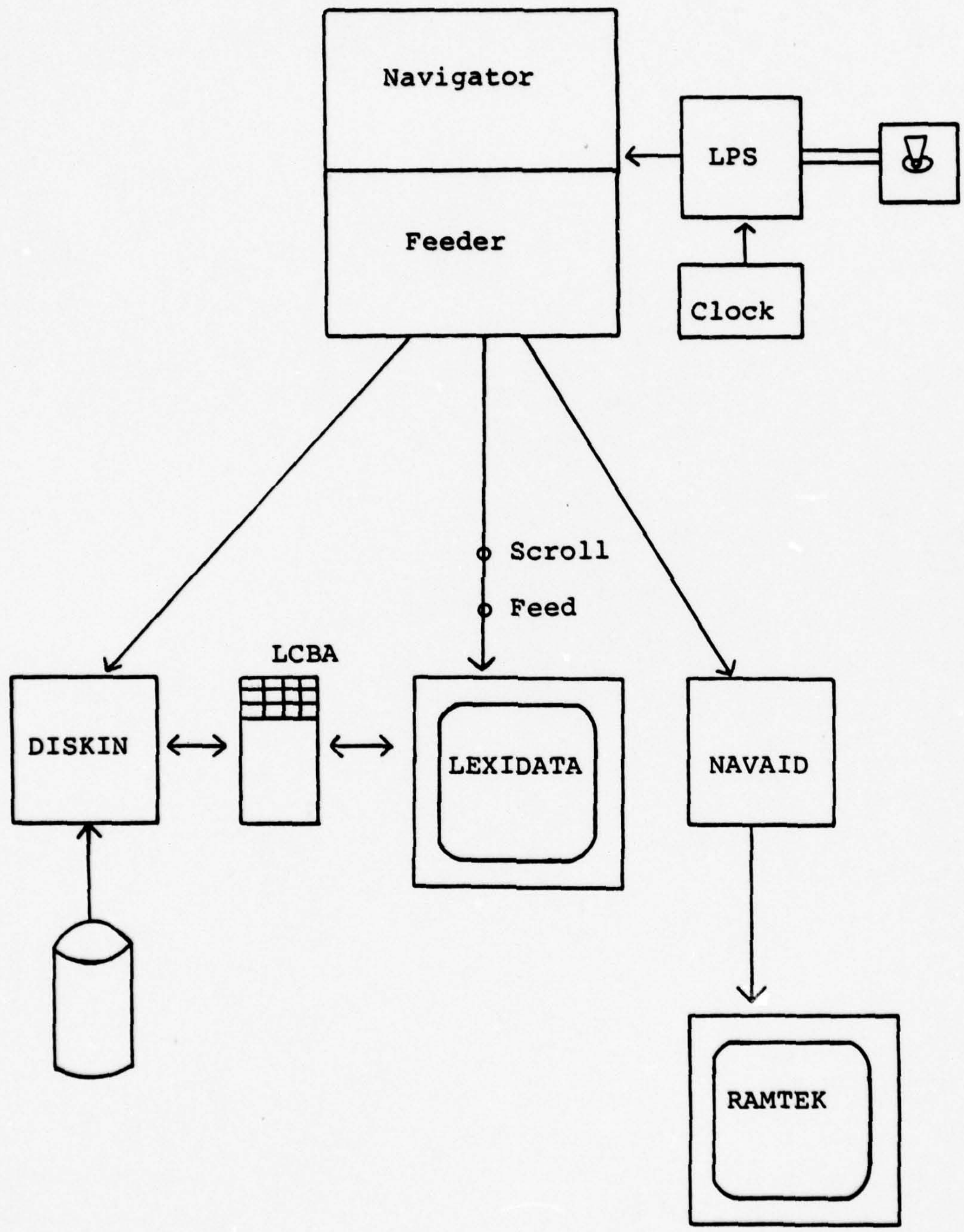
Instead, the designers of Unix encouraged the programmer to deal with such situations by use of multiple processes communicating through pipes. Unfortunately, the experience related above established that this solution was not satisfactory.

Accordingly, BBN was contracted to install a modification to the block i/o routine (physio) to allow the process calling it to initiate an i/o request, continue execution, and check at a later time for completion of the request. By restricting such use to data transfers involving the shared large core buffer area (LCBA) the danger outlined above was eliminated. This change is not complex and can easily be installed in other Unix systems.

The navigator and feeder processes were then combined into one task, as illustrated in Figure 3.6. This required making use of split instruction and data spaces, a feature not especially well supported in Unix.

A new motion system was constructed combining the navigator and the feeder into one process. This change resulted in significantly improved speed and smoothness of motion.

Motion Processes Figure 3.6



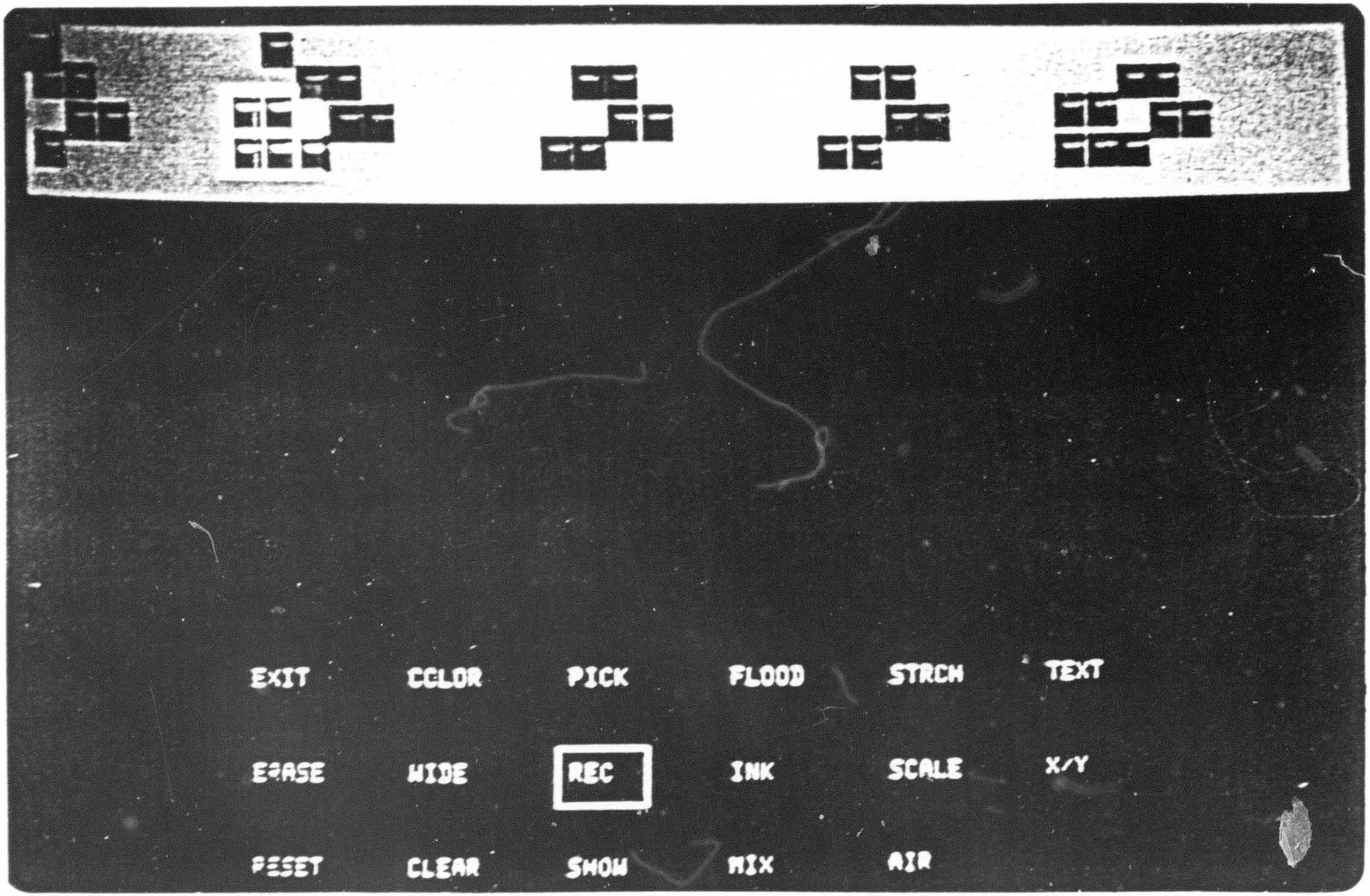


Plate 2 Navigational Aid and Menu

4. IMAGE PLANE EDITING

The prototype SDMS includes a set of tools for creating and modifying image planes and their associated navigational aids. These tools are integrated into the motion facilities so that the user can move to the location to be edited in the same manner as he would move to it to peruse the data located there. Once there, pressing the stylus to the data tablet is activates the image creation and editing facilities. The various operations are controlled by selecting from a menu displayed on the key map monitor. These operations allow the input of graphic primitives, such as lines and circles, and permit the user to modify the operation of these input operations, such as by specifying colors and widths of lines.

While the image plane editing programs bear a strong resemblence to similar programs for graphic illustration developed in laboratories at Xerox PARC, MIT, and New York Institute of Technology, the programs in SDMS are the first to offer such tools together with the ability to move continuously about a drawing surface which is arbitrarily larger than the display screen. In addition, the multiple screen capability of SDMS, by allowing the menu to be placed on a seperate screen from the image being created, permits the display of all possible menu options,

eliminating the need to choose between providing a complicated layering of commands or reducing the area of the display available for display of the painting surface.

4.1 Using the Editor

The editing program makes use of two screens. The main screen presents a view of the image plane. Pressing the pen to the tablet causes a color pallet to appear at the bottom of the screen and activates the editor.

An auxiliary screen presents a menu of operations which may be performed. Each selection is indicated by a colored rectangle containing the name of the operation which it invokes. The user indicates the desire to make a selection from the menu by moving the stylus to the left hand side of the tablet. At this point, the cursor appears on the auxiliary screen and can be positioned over the desired menu button. Pressing the stylus to the tablet enters the selected mode, which is indicated by the button being framed in color. If the selected operation requires the input of a coordinate from the image plane, the cursor moves back to the main screen and tracks the stylus there.

4.2 Commands

This section decribes each of the facilities available to the user of the image plane editor.

COLOR specifies the color to be used in subsequent operations. After touching the color button, the user points to any place on the screen to indicate the desired color. The cursor under the pallet moves to indicate the new color which was selected. This mode is used primarily to select a color which has already been used in the image. The user may choose a color from the pallet at any time without using the color button merely by touching the pen to the desired spot on the pallet.

INK allows the user to input free-hand lines and curves, much as if he were drawing on the image plane with a regular pen in the color most recently selected.

RECT accepts two points and draws a rectangle between them in the specified color. While the pen is near the tablet, a white rectangular cursor is displayed to show the location of the rectangle which will be drawn when the pen is depressed, so as to allow the user to position it exactly.

CIRC similarly allows the input of circles.

TEXT prompts the user to type in the desired text string and specify a size. The cursor then assumes a rectangular shape of the same size as the text string. When the pen is touched to the tablet, the text is entered at the specified position in the specified color.

FLOOD fills a shape with color.

STRETCH allows input of "rubber band lines" which stretch from the first digitized point to the pen as it is moved about, allowing the easy input of straight lines.

PICK allows a rectangular area to be copied from one place to another, with optional scaling.

WIDE is like INK but with a wider pen point.

AIR allows the input of texture.

4.3 Mode Control

Two additional commands alter the effect of the preceding commands.

GRID sets up a rectangular grid to which coordinate input can be forced to aid in drawing regular geometric shapes.

MIX allows the specification of new colors by mixing the three primary colors.

4.4 Implementation

As mentioned previously, the CCA SDMS is unique in allowing a user to draw on a surface much larger than the display and to scroll over that display. This feature requires that graphical operations be performed on the image plane as it is stored in the PDP-11 core buffer. Such an approach allows operations that span a width greater than that of the display. It also allows implementing features not included with the display, such as shape filling and high quality text. In addition, the same low level routines can be used by the picture construction program described in Section 5.

The GDS editor uses the tile map described in Section 3.1.1 to map the coordinates of points to be modified into core addresses. When a location is modified, its containing tile is flagged as having been updated so that it can be written back to the disk. The result of the operation is also displayed on the screen as feedback to the user, so that the display and the core buffer are always in agreement.

5. ICON CREATION

Early in the fifth quarter of the SDMS project a new feature will be demonstrated that has never before been possible with a Spatial Data Management System: the ability to use SDMS as a view of a symbolic database management system. This will be accomplished through a set of tools which can populate an I-Space with icons generated from data in a DBMS. These tools are a combination of three facilities:

1. A symbolic database management system. In this prototype, the DBMS is INGRES, a relational database developed at the University of California at Berkeley.

2. A language for specifying the appearance of an icon as a function of data in the DBMS. This Icon Class Description Language (ICDL) is interpreted once for each tuple to be displayed.

3. An interface between the DBMS and ICDL. This interface is in the form of a query language called SQUEL, which is an extension of the INGRES query language, QUEL.

5.1 INGRES

INGRES is a relational database management system developed expressly to run under Unix. In place of the files, records, and fields of conventional DBMS's, INGRES provides relations, tuples, and attributes. A tuple typically contains information about one particular entity. For example, in a database of ships, there may be a relation giving each ship's location. Within that relation, there would be a tuple for each ship, containing the name of the ship and its location.

The implementation of INGRES includes a query language, QUEL, which allows a user to enter and retrieve information. Retrieval is typically accomplished by specifying a relation and some qualification for selecting the tuples to be retrieved. For example, to retrieve all ships flying the U.S. flag, the user might type:

```
RETRIEVE SHIPS WHERE (SHIPS.COUNTRY="US")
```

More complex retrievals can be composed by combining qualifications on one or more relations and attributes within relations.

Another useful feature of INGRES is EQUEL, a facility which allows a programmer writing in C, the implementation language of Unix, to access the database. Partially parsed QUEL statements can be passed through EQUEL to

INGRES, where they are processed, passing the results back to the user program. EQUEL allows such user programs to make use of all of the power of INGRES without needing intimate knowledge of the internals of the DBMS, and aid to the implementation and portability of such programs.

5.2 SQUEL

The queries of the symbolic database management system are virtually the only typing required by SDMS. Accordingly, the SQUEL monitor program is also the command interpreter for SDMS. Expressions typed by the user are parsed by YACC [JOHNSON], a parser included as part of Unix. If the expression is a valid QUEL command, it is passed on to INGRES. If it is one of the SDMS extensions to QUEL, the SQUEL monitor will set various operations in motion, possibly including the entering of one or more QUEL requests.

In the current discussion, the most significant SQUEL statement is the ASSOCIATE statement. This statement functions very much like the RETRIEVE statement of QUEL. Rather than print the resulting tuples, however, the ASSOCIATE staement passes them off the association processor which, in turn, passes them one at a time to the icon creation module which actually interprets the ICDL and

creates the icons. The association processor also maintains a record of all icons and their corresponding tuples for use in updating the graphical data space if the INGRES database is changed. These processes are shown schematically in Figure 5.1.

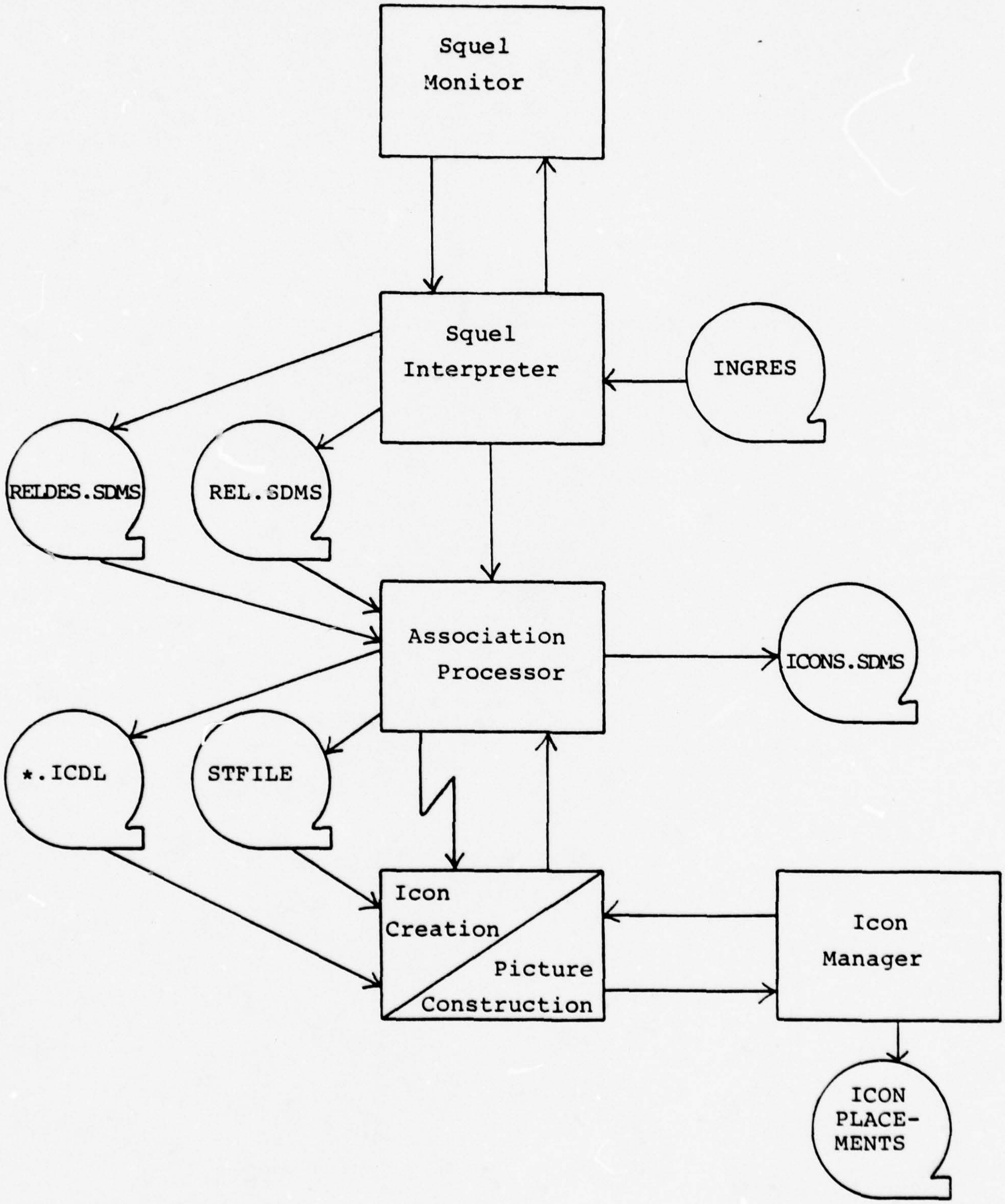
The full format of the associate statement is as follows:

```
ASSOCIATE [relation] USING [icdl_description]
 WHERE [qualification]
```

The relation and qualification are exactly as in QUEL. The USING clause allows the specification of one of a number of ICDL descriptions. The SQUEL interpreter builds a file containing the format of the relation, the selected tuples, and the name of the ICDL. It then informs the association processor via a pipe that the an association is waiting to be processed. The association processor then processes the tuples one at a time, building a symbol table for each one containing the values of the attributes of the tuple. This symbol table is then passed to the icon-creation module.

Icon Creation Processes

Figure 5.1



5.3 Icon Creation

The icon creation module actually turns the symbolic data from INGRES into a graphical representation. For each tuple supplied by the association processor, it interprets an Icon Class Description which describes how the icon should be drawn as a function of the symbolic data.

Icon Class Descriptions are typically written by the data-base administrator for use by a number of users. Such descriptions allow the shape, size, color, and text of an icon to be specified. The Icon Class Description used to generate the icons shown in Plate 3 is given below:

```
icon class shipbyclass(r) of relation ship
  begin
  maximum size is (100,100);
  position is (r.class*900+150,210);
  use picture 1
    begin
    picture 1
      begin
      image plane 0
        begin
        template icon 0;
        attribute region r.type from (10,70) to (90,80);
        attribute region r.nam from (10,85) to (90,95);
        end;
      end;
    end;
  end;
```

The relation ship contains tuples having the attributes class, type, and name.

The first line of the program specifies that this is an icon class description, named shipbyclass, which expects a tuple variable r and is to be used with the relation ship.

Following the begin the maximum size statement tells the icon manager how much space to allocate for this icon.

The position statement tells the icon manager where to try placing the icon. In this case, the integer value of the class attribute is used in an expression to divide the Information Space into regions along the x axis, such that the region in which an icon is placed is a function of its class.

The use picture statement selects the picture block from the following lines of code which has as its argument the same expression. In this particular example, there is only one picture block.

The image plane statement specifies that the following block contains information for image plane 0, the highest level image plane and the only one supported at this time.

The template icon statement specifies a template which is to be used as the background of all of the icons to be generated. It is drawn with the aid of the Graphical Data Space Editor described in the preceding section.

The two attribute region statements cause the character strings extracted from the tuple attributes type and name to be placed at the indicated locations within the icon.

As each icon is created, it is passed to the Icon Manager, a module responsible for keeping track of all icons and ensuring that none of them overlap. If the target position specified by an ICD is already occupied, the icon manager will attempt to place the icon at the nearest empty location.

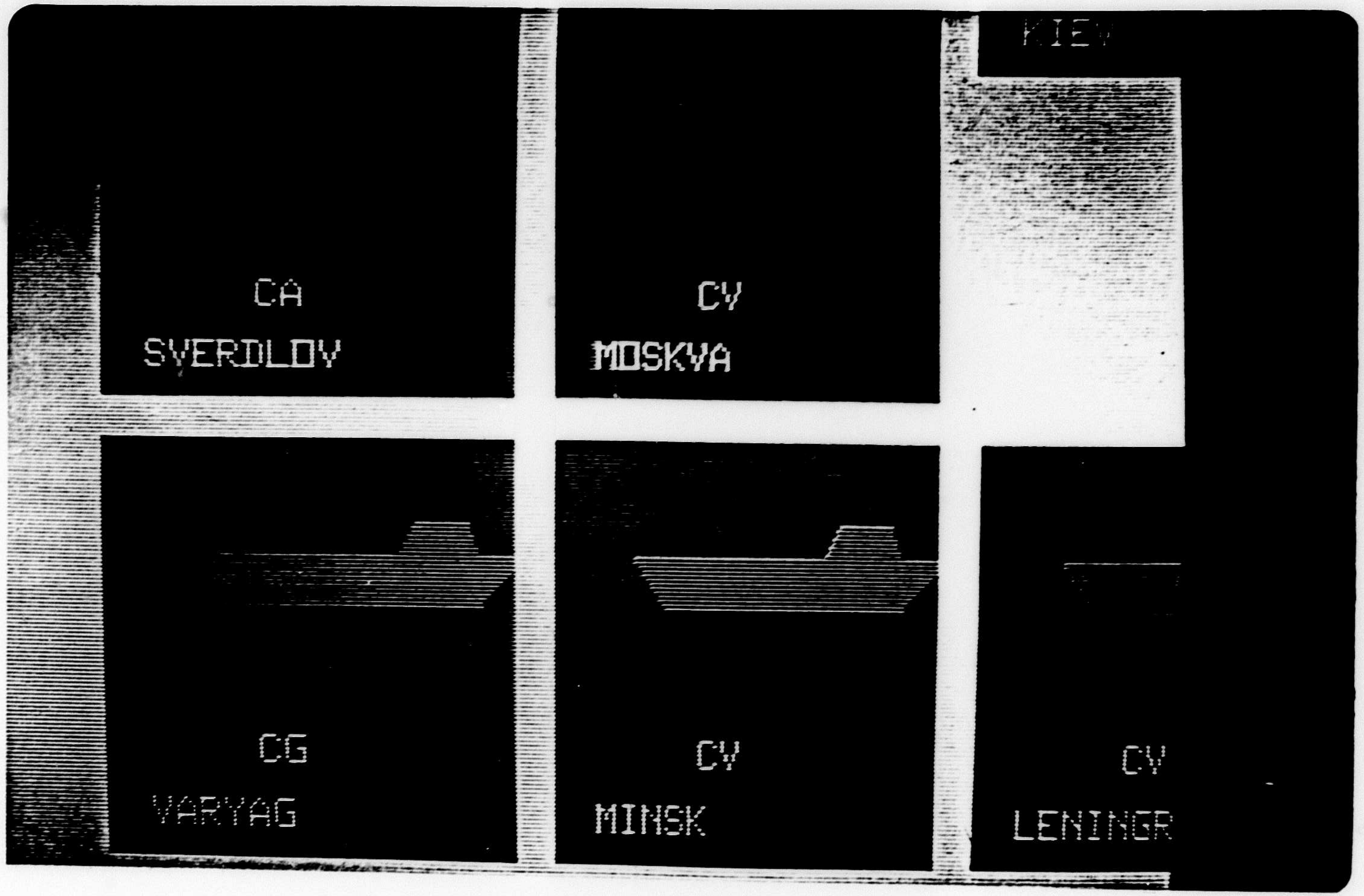


Plate 3 The Information Space

The icons were generated from the ICDL of chapter 5.

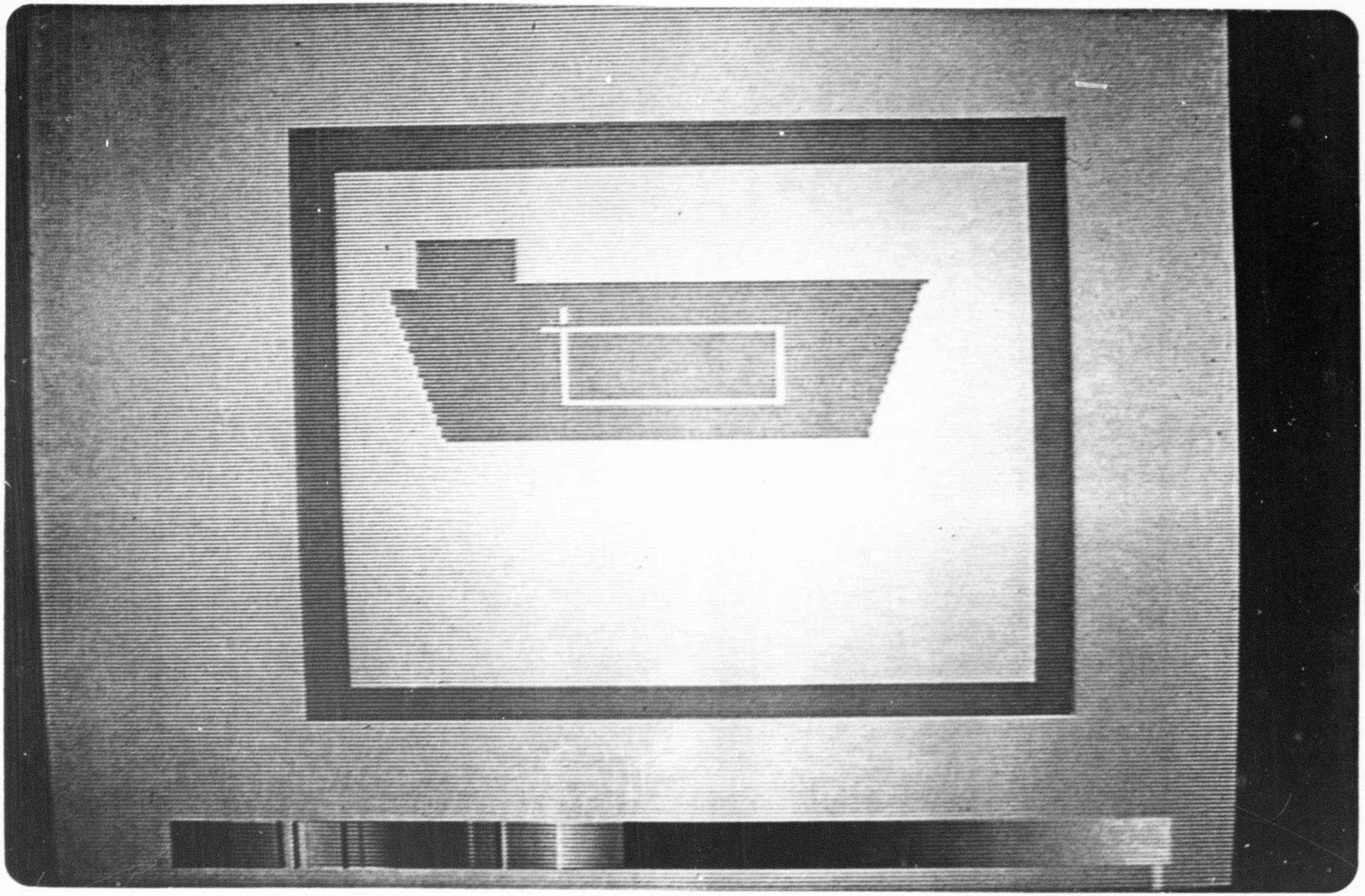


Plate 4   Icon Being Painted

White rectangle indicates position and sixe of text
which is about to be placed.

Plate 5 Completed Icon and Scaled Down Copies

Note that the icon has been partially scrolled in order to gain access to a blank area on which to place the scaled down copies.

Appendix A

NAME

add icon - adds an icon to the icon database.

SYNOPSIS

```
        /* adds an icon */
long add_icon(move_flg,coord,size,parent,source,type)

        /* adds an attribute region */
long add_attr_icon(move_flg,coord,size,parent,source,id)

        /* adds a port to a coordinate in the GDS */
long add_gds_port(move_flg,coord,size,parent,source,
                  t_coord,scale)

        /* adds a port to an icon */
long add_icon_port(move_flg,coord,size,parent,source,
                   t_icon,scale)

        /* adds a port to a UNIX process */
long add_proc_port(move_flg,coord,size,parent,source,
                   program,arg1,arg2)

int move_flg ;  /* can icon can be moved for a fit? */
struct gds_coord coord ;  /* GDS coord of icon */
struct gds_size size ;  /* extents of icon */
long parent ;   /* parent icon */
int source ;    /* class assoc or GDS editor */
int type ;      /* icon type */
int id ;        /* id for attribute region */
                /* following fields are for ports */
struct gds_coord t_coord ; /* GDS coord for target */
int scale ;     /* scale for target */
long t_icon ;   /* icon id for target */
char *program ; /* program for UNIX process port */
char *arg1 ;    /* arg1 for program */
char *arg2 ;    /* arg2 for program */

where the structures are:

struct gds_coord
  { int   I_space ;      /* I-Space id */
    float x,y ;          /* universal coord in GDS */
    int   z ;            /* (plane number) */
  } ;

struct gds_size
  { float x_ext,y_ext ; /* extents in GDS */
    int   z_ext ;        /* (number of planes) */
  } ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

These routines add icons to the database of the icon manager. If the icon fits at the requested location, the icon is added. If it does not fit, and if move_flg is true, the area nearby is searched for a place to put it. If a place is found, it is added, otherwise, the call fails.

add_icon - adds most types of icons.

add_attr_icon - adds icons of type attribute region.

add_gds_port - adds a port that have a GDS coordinate as their target.

add_icon_port - ads a port that has an icon as its target.

add_proc_port - adds a port that has a UNIX process as its target.

DIAGNOSTICS

Each of these routines returns:
 icon id - of the new icon if successful.
 -1 if unsuccessful for access or limitation reasons.
 -2 if icon will not fit.

FILES

"icons": the file holding all icon data.

"ports": the file holding port data.

I-Space directory: files which hold data for quick access to icons for a given region of an I-Space.

"deleted": file of deleted icon id numbers.

NAME

delete_icon - deletes an icon from the icon managers database.

SYNOPSIS

```
delete_icon(id)

long id ;        /* icon to delete */
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

Deletes the given icon if it exists. Any children are also deleted.

DIAGNOSTICS

Returns:
 0 if successful
-1 if icon does not exist.

FILES

"icons": the file holding all icon data.

"ports": the file holding port data.

I-Space directory: files which hold data for quick access to icons for a given region of an I-Space.

"deleted": file of deleted icon id numbers.

NAME

move_icon - used to move an icon in the database of the icon manager. in the GDS.

SYNOPSIS

```
move_icon(icon_id,move_flag,coord)

long icon_id ;  /* id of icon to move */
int move_flg ;  /* true if "nearby" coord is ok */
struct gds_coord coord ; /* target coordinate for icon */
```

where the structures are:

```
struct gds_coord
  { int   I_space ;      /* I-Space id */
    float x,y ;          /* universal coord in GDS */
    int   z ;            /* (plane number) */
  } ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

This routine finds a new position for the given icon. It should be used before the icon is actually moved in the GDS. This call reserves the space for the new location. If it is successful, picture construction routines can be called to actually move the icon.

If move_flg is false and the icon does not fit at the specified location, the call is unsuccessful. If move_flg is true and it does not fit, the surrounding region is searched for a place. If found, the call is successful.

DIAGNOSTICS

Returns:

1 if successful but not in the requested location (another call to the icon manager must be made to get the new location)

0 if successful at the requested location

-1 if icon does not exist already

-2 if space could not be found

FILES

"icons": the file holding all icon data.

I-Space directory: files which hold data for quick access to icons for a given region of an I-Space.

NAME

size - change the size of an icon.

SYNOPSIS

```
size_icon(icon_id,move_flag,size)

long icon_id ;  /* id of icon to change size */
int move_flg ;  /* true if "nearby" coord is ok */
struct gds_size size ; /* new size of icon */
```

where the structures are:

```
struct gds_size
  { float x_ext,y_ext ; /* extents in GDS */
    int   z_ext ;       /* (number of planes) */
  } ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

Updates the size of an icon. If the new size is smaller in all dimensions, the icon does not move. If it is larger and it fits in its current place, it does not move. If it does not fit and the move_flg is true, the region nearby is searched for placing the icon. If no space is found, this call fails. This routine should be called <u>before</u> the picture in the GDS is changed.

DIAGNOSTICS

Returns:

1 if successful but not in the requested location (another call to the icon manager must be made to get the new location)
0 if successful and icon is in same location
-1 if icon id is invalid
-2 if it could not fit anywhere

FILES

"icons": the file holding all icon data.

I-Space directory: files which hold data for quick access to icons for a given region of an I-Space.

NAME

upd_icon - updates the fields of an icon except for size and position

SYNOPSIS

```
            /* updates an icon */
long upd_icon(icon_id,move_flg,parent,source,type)

            /* updates an attribute region */
long upd_attr_icon(icon_id,move_flg,parent,source,id)

            /* updates a port to a coordinate in the GDS */
long upd_gds_port(icon_id,move_flg,parent,source,
                  t_coord,scale)

            /* updates a port to an icon */
long upd_icon_port(icon_id,move_flg,parent,source,
                   t_icon,scale)

            /* updates a port to a UNIX process */
long upd_proc_port(icon_id,move_flg,parent,source,
                   program,arg1,arg2)

long icon_id ;   /* id of icon to update */
int move_flg ;   /* can icon can be moved for a fit? */
long parent ;    /* parent icon */
int source ;     /* class assoc or GDS editor */
int type ;       /* icon type */
int id ;         /* id for attribute region */
*/
long t_icon ;    /* icon id for target */
char *program ;  /* program for UNIX process port */
char *arg1 ;     /* arg1 for program */
char *arg2 ;     /* arg2 for program */
```

where the structures are:

```
struct gds_coord
  { int   I_space ;      /* I-Space id */
    float x,y ;          /* universal coord in GDS */
    int   z ;            /* (plane number) */
  } ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

Updates an icon. The fields for position and size must be changed with move_icon and size_icon,

respectively. Other fields may be changed with this call.

DIAGNOSTICS

Returns:
 0 if successful
 -1 if icon id is invalid

FILES

"icons": the file holding all icon data.

"ports": the file holding port data.

NAME

get_icon - retrieves the data for an icon.

SYNOPSIS

```
get_icon(icon_id)

long icon_id ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

Retrieves the data for an icon given the icon id number.

DIAGNOSTICS

Returns:

 0 if successful, with data in shared buffer
-1 if icon does not exist

FILES

"icons": the file holding all icon data.

"ports": the file holding port data. (files)

NAME

who_point - find the icons which touch a specified point in the GDS.

SYNOPSIS

```
who_point(coord)

struct gds_coord coord ;  /* GDS coord for query */

struct gds_coord
  { int   I_space ;       /* I-Space id */
    float x,y ;           /* universal coord in GDS */
    int   z ;             /* (plane number) */
  } ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

Finds all icons which touch the given point in the GDS. Remember, if more than one touch, they must be nested icons. This routine returns a list of icons in order from highest parent to lowest child. The I-Space icon is <u>never</u> included.

ALGORITHM

The region surrounding the point in question must be loaded into core. Then search for one icon which touches that point. Then trace to the top-most parent and return it and all its descendents which touch the point.

DIAGNOSTICS

Returns:

0 always, with the icon id numbers in a shared buffer.

FILES

"icons": the file holding all icon data.

I-Space directory: files which hold data for quick access to icons for a given region of an I-Space.

NAME

who_region - returns all the icons within a region of the GDS.

SYNOPSIS

```
who_region(coord,size)

struct gds_coord coord ;  /* GDS coord for query */
struct gds_size size ;   /* size of region */

struct gds_coord
  { int   I_space ;      /* I-Space id */
    float x,y ;          /* universal coord in GDS */
    int   z ;            /* (plane number) */
  } ;

struct gds_size
  { float x_ext,y_ext ; /* extents in GDS */
    int   z_ext ;        /* (number of planes) */
  } ;
```

PROCESS MEMBERSHIP

icon_manager

DESCRIPTION

Finds all icons which touch the given region in the GDS. The icons are returned in a list which is not sorted in any way.

ALGORITHM

The specified region is loaded into core and searched. All icons touching the region are flagged. When the search is done, the id numbers of the flagged icons are put into a shared buffer.

DIAGNOSTICS

Returns:

0 always, with the icon id nnumbers in a shared buffer

FILES

"icons": the file holding all icon data.

I-Space directory: files which hold data for quick access to icons for a given region of an I-Space.

Icon Creation

The icon creation module is the only module which executes ICDL. Its purpose is to execute ICDL and to guide the creation of icons through the use of the picture construction routines.

Data structures

<u>per icon data</u>:

local id number (used for referencing sub-icons), local id number of parent icon, maximum x and y extent, target GDS coord of origin of icon,

<u>per i-plane data</u>:

source icon id, scale, orientation, (x,y) of upper-left corner of free text region, x and y extent of free text region

<u>per color statement</u>:

color, (x,y) of position for filling

<u>per text statement</u>:

text string, (x,y) of upper-left corner of text region, x and y extent of text region

<u>per update region statement</u>:

text string, update region id, (x,y) of upper-left corner region, x and y extent of region

5.4.2 Icon Creation Functions

The icon creation process falls into 4 parts:

1. Compile the icon class description, if necessary.
2. Execute the compiled file to obtain the parameters for icon construction.
3. Reserve space for the icon in the GDS.
4. Post the picture in the GDS using picture construction routines.

Each part is a separate routine in the following description.

NAME

create_icon - given a tuple, icon class description and I-Space identifier, it creates an icon for the tuple and places it in the GDS.

SYNOPSIS

create_icon (tuple,assoc_id)

```
struct tuplestr tuple ; /* the entity tuple */
char *icd ;       /* icon class to use */
int  assoc_id ; /* id for class association to use */
```

tuple is the linked tuple for the icon to be created. icd is a Unix file name where the icon class description resides. Otherwise, it is compiled first by the icdl compiler. i_space is an I-Space identifier.

PROCESS MEMBERSHIP

icon_creation

DESCRIPTION

This routine is called with three arguments, a tuple, an icon class description and an I-Space identifier. It creates an icon for the tuple from the given icon class description and places it in the given I-Space.

ALGORITHM

1. Read the parameter values into params, the icon parameter data structure.
2. Ask icon manager to reserve space for the free text and update regions.
3. For each plane in the I-Space that has an icon do the following using the picture construction primitives:

    1. Create a scratch copy of the icon for this i-plane
    2. Perform all coloring.
    3. Perform any rotation and scaling.
    4. Write any text onto the scratch picture.
    5. Place the picture in the appropriate place in the GDS.

4. log the successful icon creation with the icon id.

DIAGNOSTICS

Returns:

icon id if it successfully created and placed the icon

-1 if the icon class was bad (i.e. could not compile) with an error message in a shared buffer

-2 if the icon could not be placed (i.e. no room

in I-Space)

FILES

icd - the file containing the ICD used by this module

/tmp/icon.params - temporary file which holds the results of executing the ICDL.

log file - logs the actions of this module.

GDS Editor Functions

NAME

pick

SYNOPSIS

pick(x1,y1,x2,y2) int x1,y1,x2,y2; - universal coordinates

PROCESS MEMBERSHIP

GDS EDITOR

DESCRIPTION

The rectangle defined by the two points is copied to a temporary buffer

ALGORITHM

being designed

DIAGNOSTICS

returns -1 if either point is outside the core buffer limits.

NAME

put

SYNOPSIS

put(mode,x1,y1,x_extent,y_extent) - place the contents of the local buffer at the location specified

int mode,x1,y1,x_extent,y_extent; x1,y1 universal coordinates

PROCESS MEMBERSHIP

GDS EDITOR

DESCRIPTION

put can be invoked in either of two modes. When mode is zero, x_extent and y_extent act as maximum limits on the size of the placed object. When mode is 1, scaling is performed to fit the buffer to the target area.

ALGORITHM

```
check that target area is in the core buffer
check that local buffer has something in it
get buffer size
if(mode==0)
     if(buf size greater than target size)
                    trunciate local buffer size
                    set return flag
     copy temporary buffer to target area ( rtn feedout|feedin)
if(mode==1)
     use rtn feedin|scale|feedout
     calling scale with the ratio (target size) / (local buffer si
```

DIAGNOSTICS

Returns:

0 if successful.
-1 if target area outside core buffer
-2 if empty local buffer
-3 if trunciation occurs while placing

NAME
fill

SYNOPSIS
fill(x,y,color) - fill region pointed to by x,y with color
int x,y,color; x and y in universal coordinates

PROCESS MEMBERSHIP
GDS EDITOR

DESCRIPTION
floods all points in the closed polygon in which x,y is located

ALGORITHM
The value of the pixel at x,y is copied to a local variable. On a raster at a time basis, first up and then down the image, each line is scanned until a new pixel intensity relative to the stored one is encountered. The scanned pixel area is replaced with the new pixel intensity. At each transition point in pixel value all neighbors of the pixel are scanned to obtain a list of local areas which are candidates for flooding, these are stacked to be processed. After checking transition areas, the stack is popped and flooding continues at the new position.

DIAGNOSTICS
Returns:
- 0 if successful.
- -1 if x,y is outside of the core buffer
- -2 if area leaked to the display screen margin
- -3 if halted abnormally by an interrupt

NAME
grid

SYNOPSIS
grid(grid_width,snapping_neighborhood) -
display a grid

int grid_width, snapping_neighborhood;

PROCESS MEMBERSHIP
GDS EDITOR

DESCRIPTION
the grid function places a grid of specified width on the display to aid the user in drawing straight lines. If the snapping neighborhood is non zero vertices of lines snap to the grid if they satisfy the neighborhood condition.

ALGORITHM
A grid is placed on the screen. If the snapping neighborhood is non zero, each point drawn is checked for snapping. The algorithm used is:

```
  /* calculate dx and dy where they represent the distance
     to the closest grid vertice */

        dx = x % grid_len;
        dy = y % grid_ht;
        if(dx > grid_len/2) dx = grid_len - dx;
        if(dy > grid_ht /2) dy = grid_ht  - dy;

   /* check if dx,dy satisfy the neighborhood condition */

        if( dx < x_neighborhood && dy < y_neighborhood )

   /* snap to the closest grid vertice */

          { newx = (x / grid_len) * grid_len;
            if((x % grid_len) > grid_len/2) newx = newx +  grid_len;
            newy = (y / grid_ht) * grid_ht;
            if((y % grid_ht) > grid_ht/2) newy = newy +  grid_ht;
          }

   /* else just return the origional x,y  */

        else

          { xnew = x;
            ynew = y;
          }

        return(xnew,ynew)
```

Vertices of drawn lines are stored when in snapping mode to allow quick deletion of the original line and rapid redrawing.

DIAGNOSTICS

Returns:

0 if successful

-1 if width is greater than the screen height or length

-2 if snapping neighborhood is greater then the grid width

NAME

reserve_menu - reserve menu area

SYNOPSIS

```
menu_descriptor                                        =
reserve_menu(virtual_tablet_fp,x,y,x_extent,y_extent,*rtn)

                                  -  reserves a menu area on
                                     the menu screen
```

PROCESS MEMBERSHIP

DESCRIPTION

Reserve menu area and prepare for future calls defining active areas of the menu. This area is used as a window on initial scanning of the menu. Menu hits signal the process and pass to it a menu_key defining what selection was made. Virtual_tablet_fp is the virtual tablet file pointer obtained by opening a virtual tablet. This pointer is necessary when a process is using more then one virtual tablet. *rtn is a pointer to the routine to be invoked on a menu strike. Reserve_menu() returns a menu_descriptor used in further references.

DIAGNOSTICS

Returns:
menu_descriptor > 0 if successful
-1 if the space was not available

NAME

close_menu

SYNOPSIS

```
close_menu(menu_descripter)
                    - unallocates the menu area
                      allocated by the reserve_menu
                      command
```

PROCESS MEMBERSHIP

DESCRIPTION

Deallocates menu area and tables set by the reserve_menu command.

DIAGNOSTICS

Returns:
  0 if successful
 -1 if the menu_descriptor is invalid

NAME

set_menu

SYNOPSIS

set_menu(menu_descriptor,x,y,x_extent,y_extent,color,menu_key)

- sets the internal menu table

PROCESS MEMBERSHIP

DESCRIPTION

Sets the internal menu table to signal the process when a menu hit occurs. The process is passed the menu_key which specifies which menu entry was picked. If color is less than 0 color is unchanged.

ALGORITHM

DIAGNOSTICS

Returns:
 0 if successful
 -1 if an invalid menu descriptor.
 -2 if defined space extends outside virtual menu

NAME

icon_menu

SYNOPSIS

icon_menu(menu_descriptor,x,y,x_extent,y_extent,icon_file_name,menu_key

- sets the internal menu table using the icon file specified.

DESCRIPTION

Sets the internal menu table to signal the process when a menu hit occurs. The process is passed the menu_key which specifies which menu entry was picked. The Icon file is read in and used as a menu picture.

DIAGNOSTICS

Returns:
 0 if successful
 -1 if an invalid menu descriptor.
 -2 if defined space extends outside virtual menu.
 -3 if the icon file can't be found or is read protected.

NAME

color_menu

SYNOPSIS

```
color_menu(menu_descriptor,x,y,x_extent,y_extent,color)
                                   - color rectangular
                                     area of menu specified.
```

PROCESS MEMBERSHIP

DESCRIPTION

Colors the menu area specified.

DIAGNOSTICS

Returns:
 0 if successful.
 -1 if an invalid menu descriptor.
 -2 if defined space extends outside virtual menu.

NAME

disable_menu

SYNOPSIS

disable_menu(menu_descriptor) - temporarily disable menu area

DESCRIPTION

Temporarily disables menu area. This routine and the one following are designed for temporarily targeting the active menu area, avoiding accidents from unappropiate menu commands and avoids dropping the menu visually. Useful when initially setting up the menu.

DIAGNOSTICS

Returns:
 0 if sucessful.
-1 if an invalid menu descriptor.

NAME

enable_menu

SYNOPSIS

enable_menu(menu_descriptor) - enable menu area

DESCRIPTION

enables menu area referenced by the menu descriptor.

DIAGNOSTICS

Returns:
0 if successful
-1 if an invalid menu descriptor.

Appendix B

NAME

template icon

SYNOPSIS

TEMPLATE ICON number

DESCRIPTION

Selects the template icon from the template image plane.

NAME
attribute region statement - defines an attribute region.

SYNOPSIS
ATTRIBUTE REGION attribute_name FROM (x1,y1) TO (x2,y2)

DESCRIPTION
Defines an attribute region and specifies its position within the icon. The region will display the value of the specified attribute.

DEFINITIONS
```
<attribute> ::=
        ATTRIBUTE REGION<attribute_name>
          FROM (<arith_expr>,<arith_expr>)
          TO (<arith_expr>,<arith_expr>) ;

<attribute_name> ::= <identifier> ;
```

EXAMPLE
```
/* attribute region for persons phone */

    attribute region r.phone from (0,0) to (50,20)
```

SEE ALSO
general(icdl), plane(icdl), example(icdl)

DIAGNOSTICS

BUGS

NAME

image plane statement - statement to describe one plane of an icon.

SYNOPSIS

IMAGE PLANE plane# plane_stmts END

DESCRIPTION

The image plane statement fully describes one plane of an icon for an icon class description. The plane# states which plane in the I-Space is intended. The plane_stmts further modify this picture. They include text, coloring, drawing pictures, etc.

The *icon used for a plane should* be one plane deep. If the icon covers more than one plane, only the top-most plane is used.

DEFINITIONS

```
<plane> ::= IMAGE PLANE <plane#> <icon_id> <plane_stmts> END ;

<plane#> ::= <arith_expr> ;

<plane_stmts> ::= <plane_stmt> |
                  <plane_stmt> <plane_stmts> ;

<plane_stmt> ::=  | <attribute region> |
```

EXAMPLE

see example(icdl)

SEE ALSO

general(icdl), icon(icdl), example(icdl)

NAME

position - defines target position for icon.

SYNOPSIS

POSITION (x,y)

DESCRIPTION

Defines the target position for each icon. The target position is the position in the I-Space where SDMS will attempt to place the created icon. This position is in the user's coordinates for the I-space. If it cannot place it at the target position, it attempts to find a position close by. If the icon cannot fit anywhere, an error occurs.

DEFINITIONS

<position> ::= POSITION (<arith_expr>,<arith_expr>) ;

EXAMPLE

```
/* set icon origin to be I-Space origin
   (taken from example(icdl))            */

   position (0,0)
```

SEE ALSO

general(icdl), icon class(icdl), example(icdl)

DIAGNOSTICS

BUGS

References

[HEROT et al.]
Herot, C.F.; Schmolze, J.; Carling, R.; Friedell, M.; Farrell, J., Detail Design Document for a Spatial Data Management System, Computer Corporation of America, 575 Technology Square, Cambridge Mass., October 6, 1978.

[JOHNSON]
Johnson, Stephen C., "YACC -- Yet Another Compiler-Compiler," in Documents for Use with the Unix Time-Sharing System, Sixth Edition, Bell Telephone Laboratories, May, 1975.